Algorithmica (1996) 15: 104–125

Algorithmica
© 1996 Springer-Verlag New York Inc.

Parallel Algorithms for Arrangements[1]

R. Anderson,[2] P. Beame,[2] and E. Brisson[2]

Abstract. We give the first efficient parallel algorithms for solving the arrangement problem. We give a deterministic algorithm for the CREW PRAM which runs in nearly optimal bounds of $O(\log n \log^* n)$ time and $n^2/\log n$ processors. We generalize this to obtain an $O(\log n \log^* n)$-time algorithm using $n^d/\log n$ processors for solving the problem in d dimensions. We also give a randomized algorithm for the EREW PRAM that constructs an arrangement of n lines on-line, in which each insertion is done in optimal $O(\log n)$ time using $n/\log n$ processors. Our algorithms develop new parallel data structures and new methods for traversing an arrangement.

Key Words. Parallel algorithms, Computational geometry, Arrangement problem, Incremental algorithms.

1. Introduction. The problem of determining the geometric structure of the intersections of curves and surfaces has a long history in mathematics [4], [6], [19]. For the purposes of computational geometry, a very important special case is that of determining this structure when the curves and surfaces being intersected are lines in $\mathbb{R}^2$ or, more generally, hyperplanes in $\mathbb{R}^d$ for $d \geq 2$. In this context the problem is known as the *arrangement* problem.

A simple and elegant sequential algorithm for computing arrangements in $\mathbb{R}^2$ was found by Chazelle *et al*. [8] and Edelsbrunner *et al*. [12]; the latter also showed how the algorithm can be generalized to $\mathbb{R}^d$. In $\mathbb{R}^2$ this algorithm has a worst-case running time of $O(n^2)$, which is obtained by inserting the lines one after another into the arrangement produced so far. In $\mathbb{R}^d$ the algorithm runs in $O(n^d)$ time. Since the problem generally requires that $\Omega(n^d)$ values be produced, the output requirements alone show that this is optimal.

Computing arrangements is an important building block in several computational geometry algorithms. In two dimensions, arrangements are used during a preprocessing step in algorithms for computing visibility graphs. They are also used by algorithms for finding shortest paths that avoid polygonal obstacles. Furthermore, the worst-case optimal hidden surface removal algorithm of McKenna [17] first projects the three-dimensional problem (involving planes) onto a two-dimensional image plane, then computes the two-dimensional arrangement produced in the image plane, and finally simplifies it to produce the viewed image.

There is a substantial body of work on the subject of parallel algorithms for computa-

[1] This work was supported by the National Science Foundation, under Grants CCR-8657562 and CCR-8858799, NSF/DARPA under Grant CCR-8907960, and Digital Equipment Corporation. A preliminary version of this paper appeared at the Second Annual ACM Symposium on Parallel Algorithms and Architectures [3].

[2] Department of Computer Science and Engineering, FR-35, University of Washington, Seattle, WA 98195, USA.

Received July 16, 1993; revised February 22, 1994. Communicated by M. Snir.

tional geometry (e.g., [1], [5], [13], and [18]). Included in this work are parallel algorithms for some problems related to finding arrangements, such as computing visibility from a point in two dimensions [5] and hidden surface removal in restricted three-dimensional scenes [18]. However, finding an optimal parallel algorithm for computing arrangements has received less attention [2], [13].

A fairly straightforward parallel algorithm for computing arrangements can be constructed using $n^2/\log n$ processors, requiring $\Theta(\log^2 n)$ time. In [13], which was the starting point for this research, Goodrich gives an output-sensitive algorithm for computing the intersections of line segments; however, when used to find arrangements of lines, its running time is no better than that of the straightforward algorithm.

We present two algorithms for the arrangement problem. The first is a deterministic algorithm for the CREW PRAM which runs in near-optimal $O(\log n \log^* n)$ time using $O(n^2/\log n)$ processors for computing arrangements in $\mathbb{R}^2$. We also show how this generalizes to an $O(\log n \log^* n)$-time algorithm using $O(n^d/\log n)$ processors in $\mathbb{R}^d$. The second solves the on-line version of the arrangement problem, in which lines are only available as input one after another. It is a randomized algorithm for the EREW PRAM that constructs an arrangement of n lines on-line, so that each insertion is done in optimal $O(\log n)$ time using $n/\log n$ processors. Both of our algorithms develop new methods for traversing an arrangement efficiently in parallel.

Perhaps because of their perceived sequential nature, very little study has been made of parallel algorithms for on-line problems. However, efficient on-line parallel algorithms can be useful in a context where extremely fast response times are required in a dynamic environment. On-line problems place unique demands on parallel algorithms because, unlike static problems, they can require efficient maintenance of data structures significantly larger than the number of processors available. In our on-line algorithm for computing arrangements we encountered a problem apparently requiring sophisticated data structures developed for sequential computation. We develop simpler data structures that are sufficient for the demands of our algorithm.

Independently of this work, Hagerup *et al.* [15] have developed a randomized CRCW PRAM algorithm for the d-dimensional arrangements problem (not on-line) that is complementary to our results and uses a different approach. At the expense of randomness they obtain an $n^d/\log n$ processor algorithm that runs in optimal $O(\log n)$ parallel time. Subsequent to the initial appearance of our result, Goodrich [14] presented a more elaborate algorithm than ours, which achieves optimal deterministic performance.

2. Background

2.1. *Problem Statement.* Given a set H of n hyperplanes in $\mathbb{R}^d$, where $d \geq 2$, their arrangement $A(H)$ is the subdivision of $\mathbb{R}^d$ they create. That is, if $H = \{h_1, \ldots, h_n\}$, and h_i^- and h_i^+ are the open half-spaces defined by h_i, then the *faces* of $A(H)$ are $\{\bigcap_{i=1}^n \tilde{h})_i$: $\tilde{h} = h_i^-, h_i$, or $h_i^+\}$. A description of an arrangement must include an enumeration of the faces, along with their topological relationships, for instance, an incidence graph. If the input hyperplanes are in general position, so that the intersection of any k hyperplanes is a $(d–k)$-dimensional face, then $A(H)$ is *simple*. The number of k-faces in a general arrangement is $O(n^d)$, and the number of k-faces in a simple arrangement is

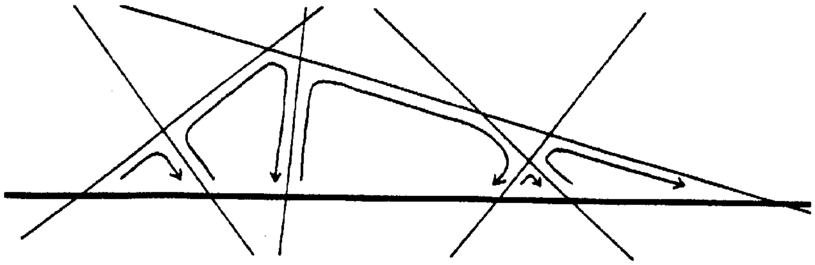

Fig. 1. Traversal during sequential line insertion.

$\Theta(n^d)$. In the two-dimensional case the points, edges, and regions of $A(H)$ are denoted by $P(H)$, $E(H)$, and $R(H)$, respectively.

We assume that the input set of hyperplanes forms a simple arrangement, and in the two-dimensional case contains no horizontal or vertical lines. The latter assumption may be eliminated by making a small rotation of coordinates if the input includes horizontal or vertical lines.

For output we need to give a description of the arrangement. In the two-dimensional case we produce, for each line in H, a sorted list of its intersections with the other lines of H. The incidence graph may be produced within the same processor and time bounds. In higher dimensions the incidence graph is produced as output.

2.2. *The Sequential Algorithm.* In $\mathbb{R}^2$ the arrangement problem can be solved by brute force in $O(n^2 \log n)$ time by computing all the intersections along each line and then sorting these n lists independently. The optimal sequential algorithm for the arrangement problem in $\mathbb{R}^2$ given in [8] and [12] removes the $\log n$ factor, using an on-line algorithm which inserts each line l into the existing arrangement of up to n lines in time $O(n)$, to achieve its running time.

For the purposes of illustration, view the line l to be inserted as being horizontal. The leftmost intersection of l with the arrangement is found and l is inserted in the list of the line that it intersects. Then a left-to-right traversal of the arrangement is made along l which discovers and adds each intersection point involving l. Given any intersection point p on l, let R be the region which l intersects immediately to the right of p. The next intersection is found by traversing the portion of the boundary of R lying above l by following the chain of edges incident to the boundary in clockwise order (this ordering is extended to infinite faces in the obvious way). Figure 1 gives an illustration of the traversal. Although it is not immediately obvious, it can be shown that such a traversal never encounters more than $3n$ segments along the way and thus the time for the insertion is $O(n)$.

3. Deterministic Algorithm for Arrangements

3.1. *Overview.* There are two key elements to the deterministic parallel algorithm. The first is the fast insertion of a single line into an arrangement, and the second is

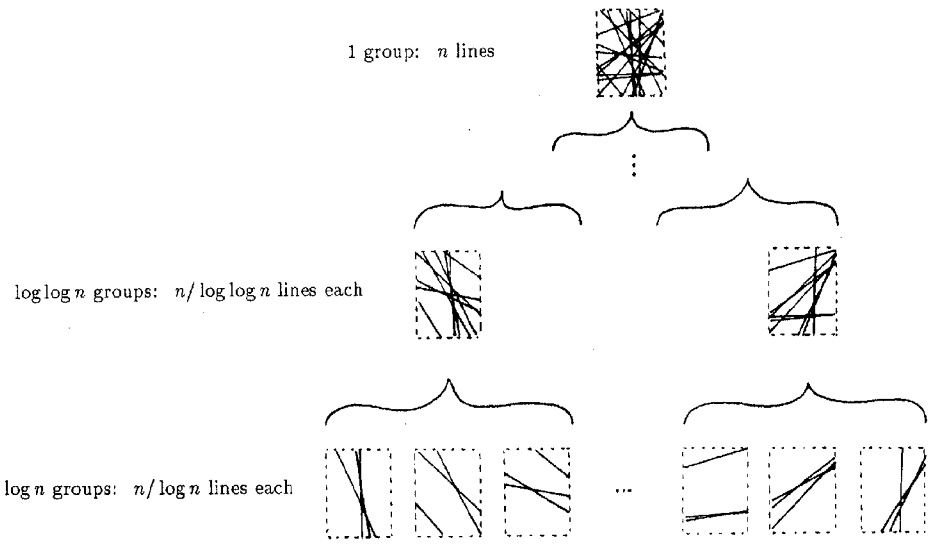
1 group: n lines
$\log\log n$ groups: $n/\log\log n$ lines each
$\log n$ groups: $n/\log n$ lines each

Fig. 2. Overview of the deterministic algorithm, in two dimensions.

the fast merging of arrangements. The insert is a parallelization of the sequential insert, which requires doing the inserts in a particular order and maintaining two extra data structures, to allow an even distribution of processors and to speed up the traversal of region boundaries. The merge of a set of arrangements is done by simultaneously inserting every line into every arrangement other than its own, and then merging the results of these separate inserts independently for each line. Somewhat surprisingly, to obtain the most efficient algorithm, we must slow down the rate at which geometric information is produced, as the bottleneck in our algorithm is the standard merging of sorted lists.

The set of input lines is denoted by H_{in}. Let $H \subseteq H_{\text{in}}$. To **insert** a line l into H means creating a sorted list of the intersection points between l and the lines in H (excluding l, if $l \in H$). If every line in H has been inserted into H, then H taken with its lines' sorted lists is called a **subarrangement** of $A(H_{\text{in}})$.

The algorithm is a divide-and-conquer algorithm which first performs a "setup step" followed by $\log^* n$ "phases." (Figure 2 gives a visual presentation of the algorithm.) The setup step orders the input lines by their slopes, and then organizes them into $\log n$ groups of $n/\log n$ consecutive lines (taken in this slope ordering). Each line is then inserted into the group which contains it. Thus the setup step provides $\log n$ disjoint subarrangements, each of size $n/\log n$.

Each phase takes as input a partition of H_{in} into k disjoint subarrangements of size n/k (in this section it will always be the case that $k \leq \log n$). A phase runs in three steps. The first step divides the input into groups of $k/\log k$ consecutive subarrangements (in the ordering of the lines they contain). It also computes two auxiliary data structures, "splitters" and "level" (which are defined later) for each of the input subarrangements.

Each line appears within exactly one subarrangement, and so appears in exactly one group, which we call the line's group. The second step inserts every line into each of the

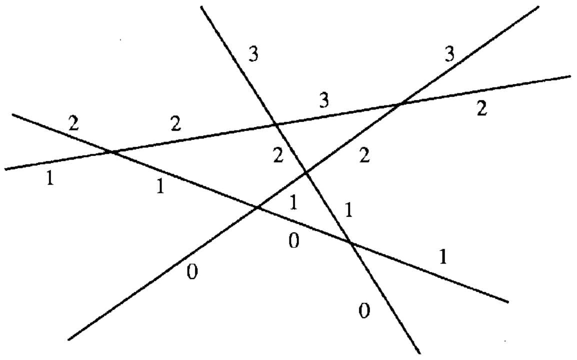

Fig. 3. Example of vertical levels.

subarrangements within the line's group. Thus $k/\log k$ sorted lists are created for every line.

In the third step the sorted lists for each line are merges into one sorted list. By creating this merged list, the line has now been inserted into its group. Thus the output is a partition of H_{in} into $\log k$ disjoint subarrangements of size $n/\log k$.

We prove, in four lemmas, that the setup step and each of a merge-phase's steps can be performed in time $O(\log n)$ using $n^2/\log n$ processors on a CREW PRAM, thus proving the main theorem of this section:

THEOREM 3.1. *Given a set H of n lines in the plane, the deterministic algorithm outlined above constructs $A(H)$ in time $O(\log n \log^* n)$ using $n^2/\log n$ processors.*

3.2. *Levels and Slope Ordering.* The key to the insertion of a line into a subarrangement is the parallelization of the sequential traversal described earlier. This is accomplished by distributing the available processors evenly along the line being inserted, in particular by assigning a processor to every $\log n$th intersection point. In order to show that our method works, we consider the level structure of the arrangement. For technical reasons we consider both horizontal and vertical levels.

If e is an edge in $E(H)$, choose a vertical line through e which does not contain any points of $P(H)$. Define the **vertical level of** e **in** $A(H)$ to be to be the number of edges of $E(H)$ this line intersects below e. It is easy to check that this is well defined (in particular, is independent of the choice of vertical line), given the fact that there are no vertical lines in the input. The set of all edges in $E(H)$ whose level is k is called the k-**level** of $A(H)$. Given any line l, define the **intersection of** l **with level** k to be the edge in level k which intersects l. Horizontal levels are defined similarly. (See Figure 3.)

The key to making our method work is the following observation about the levels of an arrangement: given a set of lines of "consecutive" slope, the level structure of their arrangement is the same for all lines whose slope lies "outside" of their set of slopes. The first step of a phase builds vertical and horizontal levels. Then the intersection of a line with the $\log n$th level, in one of these two directions, can be computed, which gives the $\log n$th intersection point. This is described in further detail below.

Each input line l of H_{in} has a slope $-\infty < m_l < \infty$. If H is a subarrangement, define $m_H^- = \min_{l\in H} m_l$ and $m_H^+ = \max_{l\in H} m_l$. Let l be a line of H_{in} such that $m_l \notin [m_H^-, m_H^+]$. If $|m_l| > |m_H^-|$ we say that l is **vertically insertable** into H, and if $|m_l| < |m_H^+|$ we say that l is **horizontally insertable** into H.

OBSERVATION 3.2. *If l is vertically insertable into H, then, with increasing y, it intersects the vertical levels in strictly increasing order.*

PROOF. For any region of $A(H)$, classify its edges as **bottom** edges if they lie below it, and **top** edges if they lie above it. Observe that all bottom edges of a region are of the same level k, for some k, and all top edges of that region have the same level $k + 1$. By the definition of insertability, if l intersects a region, then it intersects the bottom of the region and the top of the region exactly once each. $\square$

OBSERVATION 3.3. *Given an arrangement $A(H)$ of n lines, a data structure can be built in $O(\log n)$ time by $n^2 \log n$ processors, which allows finding the intersection of a vertically insertable line with any level of $A(H)$ in time $O(\log n)$ by a single processor.*

PROOF. An ordering is defined on all edges in the arrangement, first by level, then within levels from left to right. A binary tree is built, with the edges as leaves, in time $O(\log n)$ using $n^2/\log n$ processors. $\square$

The similar observations hold for horizontally insertable lines for increasing x and horizontal levels.

3.3. *The Algorithm*

3.3.1. *Setup Step*. Ordering the n input lines by the their slopes is done as a sort of n scalar in time $O(\log n)$ by n processors. Breaking the lines into groups of $n/\log n$ lines can be done in constant time by n processors. Each line must now be inserted into a group of $n/\log n$ lines. Assign $n/\log n$ processors to each line. For a specific line l, each of its processors finds the intersection of l with a different line in l's group. To sort these intersection points is done as a sort of $n/\log n$ scalars by $n/\log n$ processors in time $O(\log n)$. This gives the following lemma:

LEMMA 3.4. *The setup step can be done in time $O(\log n)$ using $n^2/\log n$ processors.*

3.3.2. *Auxiliary Data Structures*. We now define splitters, which facilitate the fast traversal of large regions (those with many edges). The use of splitters first appears in [7]; it was also used in [13]. If e is an edge of $A(H)$, let R be the region below e. The **splitter for e in** $A(H)$ is the rightmost edge among the bottom edges of R. (See Figure 4.) If R has no bottom edges, then the splitter is undefined for the edges of R. This occurs only for the “bottommost” region, which will not be traversed.

We describe how to attach a pointer from every edge to its splitter, in a subarrangement of n/k lines, in time $O(\log n)$ using $n^2/(k \log n)$ processors. Note that such a subarrangement has $(n/k)^2$ edges, so there are $\log n/k$ edges per processor. To begin, every edge

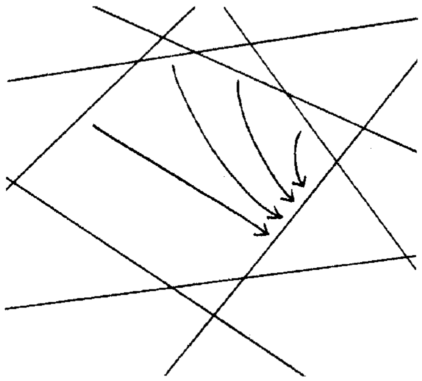

Fig. 4. Pointers to (downward) vertical splitters.

(except one) sets a pointer to its clockwise neighbor in the region below itself. The exception is the right-hand infinite edge of the bottommost region, which points to itself. Thus there is at most one list for every region, whose root is that region's splitter (except in the case of the bottommost region). The goal is to have every edge point to the root of its list. By using list-ranking, we can set each edge's pointer to the root of its list within the desired time bounds. Note that the edges of the bottommost region will all be pointing at that region's right-hand infinite edge.

To calculate levels, observe that the level of any edge is one greater than that of its splitter, except in the case of edges of the bottommost region, which all have level 0. The right-hand infinite edge of the bottommost edge will be the **root edge** for the purposes of this step. The starting configuration for making levels is just the result of the construction of splitters. These pointers are now labeled with 1, unless they point at the root edge, in which case they are labeled with 0. This gives a tree whose root is the root edge. Again using list-ranking, along with an Eulerian tour, we can compute the cost of the path from each edge along these pointers to the root edge in the desired time bounds, which is exactly the level of the edge.

LEMMA 3.5. *Given a subarrangement of n/k lines, where $k \leq \log n$, its splitters and levels can be produced in $O(\log n)$ time using $n^2/(k \log n)$ processors.*

3.3.3. *Inserting a Line into a Subarrangement.* As a result of the ordering done in the setup step and by merging of consecutive subarrangements, whenever we do an insert the line will be either vertically or horizontally insertable. A **vertical insert** or a **horizontal insert** will be done in each case, respectively. We describe the vertical insert of a line l into a subarrangement $A(H)$ of n/k lines, using $n/(k \log n)$ processors; the horizontal insert is similar.

A vertical insert is done in two passes, called **traversals**, the first downward and the second upward; we describe the downward pass. A subarrangement of n/k lines has n/k levels. Assign a processor to every $\log n$th level, and also to level n/k. Subscript the processors by successive positive integers in order of the levels to which they are assigned. Each processor P_i first finds the intersection e_i of l with its level, and computes

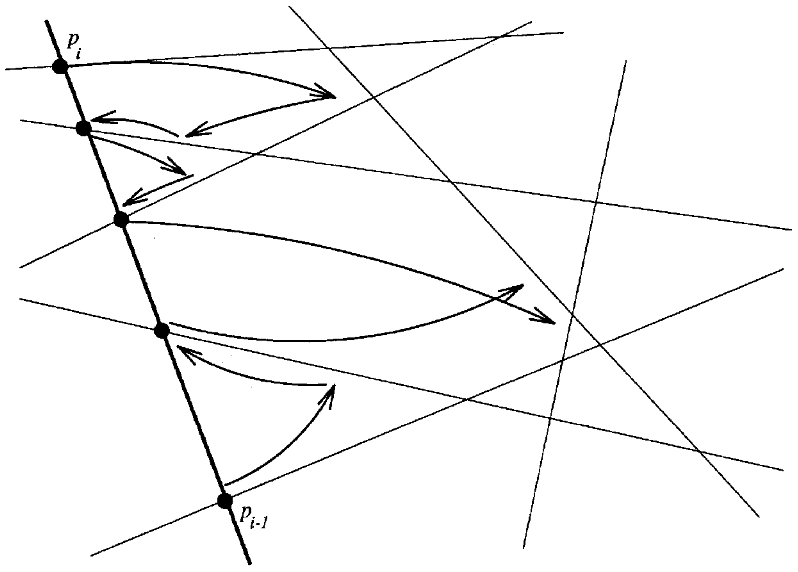

Fig. 5. Example of downward and upward traversals.

the intersection p_i of l and the line containing e_i. Let R be the region below e_i. The processor now begins a clockwise traversal of the boundary of R. This cannot actually be done edge-by-edge, as it would take too long for large regions. Instead, the processor immediately jumps to its splitter, and then the clockwise search proceeds as it would in the sequential case. If this traversal reaches an edge e' which intersects l, the process is started over, and so on. The processor stops when it reaches p_{i-1} or encounters an edge whose containing line intersects l below p_{i-1}.

The upward pass is now done, traversing boundaries counterclockwise, using the appropriate redefinition of splitters, etc. (See Figure 5.) What needs to be proven is that this takes time $O(\log n)$, and that each intersection point of l with the lines of H is found by either the downward or upward pass. This is enough to give the sorted order of the intersection points.

LEMMA 3.6. *The above algorithm inserts a line into a subarrangement of size n/k, in time $O(\log n)$ using $n/(k \log n)$ processors.*

PROOF. The allocation of processors takes constant time. Consider processor P_i, where $1 \leq i \leq n/k$. For P_i to find the intersection of l with its level takes time $O(\log n)$, by Observation 3.3. The following argument is due to Goodrich [13].

Divide the set H of lines H_{above}, H_{between}, and H_{below}, as the lines intersect l above p_i, between p_i and p_{i-1} inclusive, or below p_{i-1}. Note that in the downward traversal, edges whose containing lines are in H_{above} are never encountered (they are jumped over), and at most one edge whose containing line is in H_{below} is encountered (then the processor stops its traversal). The cardinality of H_{between} is less than $\log n$, by the allocation of the processors. Consider the standard sequential traversal which would be performed when inserting l into $A(H_{\text{between}})$. The number of edges encountered in such a traversal is $O(\log n)$. The number of edges encountered in our traversal is no greater than the number in such a sequential traversal, hence is $O(\log n)$. Thus the traversal takes time $O(\log n)$.

If the downward traversal by processor P_i reaches p_{i-1}, or the upward traversal by P_{i-1} reaches p_i, then all the intersection points between p_i and p_{i-1} have been found. If the upward and downward traversals each terminate by reaching an edge contained in H_{above} and H_{below}, respectively, then make the following argument. Consider the region formed by the lines of $H_{\text{above}} \cup H_{\text{below}}$ which contains p_i and p_{i-1}. If the boundary of this region is followed from top to bottom in the clockwise direction, starting from l, then the edges encountered will first all be contained in lines of H_{above}, then all contained in lines of H_{below}. This is also true in our traversal, so in the case under consideration the two traversals will have "passed" each other. Thus all intersection points have been found. □

3.3.4. *Merging Sorted Lists*. In the third step of each phase, every line must merge $k/\log k$ sorted lists using the processors assigned to it:

LEMMA 3.7. *Let $k \leq \log n$. $k/\log k$ sorted lists of length n/k can be merged in time $O(\log n)$ using $n/\log n$ processors.*

PROOF. A balanced binary tree is formed with the lists at the leaves. The lists are merged in rounds, so that each round reduces the depth of the tree by one. Thus there are $\log k$ rounds. Each round can be completed in $O(\log n/\log k)$ time using $n/\log n$ processors. This is done by slowing down the optimal work $O(\log\log n)$ merging algorithm [16]. □

3.4. *Higher-Dimensional Arrangements*. Given a set H of n hyperplanes in $\mathbb{R}^d$, the d-dimensional arrangement $A(H)$ is the subdivision of $\mathbb{R}^d$ generated by H. An optimal worst-case sequential algorithm for constructing $A(H)$ is given in [12], which runs in $O(n^d)$ time. We show that our two-dimensional arrangement algorithm can be used to solve the d-dimensional problem in $O(\log n \log^* n)$ time using $n^d/\log n$ processors (thus is within $O(\log^* n)$ of optimal). The algorithm begins by computing the projection of the problem onto each of the two-dimensional planes formed by the intersection of $d-2$ hyperplanes. Each of these two-dimensional arrangement problems is solved, and then the results are combined to build the higher-dimensional structure of the d-dimensional arrangement. The combining process takes $O(\log n)$ time with $n^d/\log n$ processors, so if a faster two-dimensional algorithm is used [14], [15], the run time of the d-dimensional algorithm is also improved.

In representing the solution to the arrangement problem, we need to represent the topological structure of the arrangement. In d dimensions an arrangement consists of k-faces, for $k = 0$ through d. (A k-face is a k-dimensional region bounded by hyperplanes of H.) We assume that the arrangement is in general position so that the intersection of k hyperplanes is a $(d-k)$-flat. The standard representation of the topological structure is the incidence graph. The incidence graph has a vertex for each face, and an edge between a k-face and a $(k+1)$-face if the k-face is contained in the $(k+1)$-face. A detailed discussion of the structure of arrangements can be found in [11]. A vertex in the arrangement is the intersection of d hyperplanes. It is important later that an arrangement vertex is incident to a constant number of k-faces.

We begin our construction of the arrangement in d dimensions by computing the intersection of each set of $d - 2$ hyperplanes. Each one of these intersections gives a two-dimensional plane. For each of these planes we compute the intersection with the remaining hyperplanes and solve the two-dimensional arrangement problem. There are $\binom{n}{d-2}$ problems, so we allocate $O(n^2/\log n)$ processors to each problem. This allows us to compute the incidence graph for all of the zero- and one-dimensional faces. If v and w are vertices of the arrangement, we say they are neighbors if they are incident to a common one-dimensional face.

We begin by creating records for the faces of the arrangement. Each vertex v creates a set of records for all of the faces incident to it. Each face is represented by a number of records. The key step in the algorithm is to pick a canonical representative for each face. We do this by attaching each face to the lexicographically minimal (lexmin) vertex that it is incident to. We now describe our algorithm for identifying the lexmin vertex adjacent to each face. Each vertex has a set of records of the form (v, f), where v is the vertex and f is a face. We create a directed graph on these ordered pairs which has outdegree at most one. If v is adjacent to the face f, the processor associated with v chooses a vertex w such that w is a neighbor of v, $w <_{\text{lex}} v$, and w is adjacent to f. The processor creates an edge from (v, f) to (w, f). If there is no such vertex, then v is the lexmin vertex adjacent to f. After constructing this graph, we identify the lexmin vertex adjacent to each face by traversing each of these trees to its root. (This can be done by an Eulerian tour and list-ranking. In order to set up the Eulerian tour, we use the fact the each vertex has a bounded number of neighbors.) After we have identified a canonical vertex for every face, we can construct the edge lists for the intersection graph. If the k-face f_k is adjacent to the $(k + 1)$-face f_{k+1}, the canonical vertex for f_k identifies this adjacency. We then do another list traversal to bring together all of the k-faces adjacent to f_{k+1}. This traversal is done using the same links as were set up to identify the lexmin vertices associated with the faces. Since list ranking can be done in $O(\log n)$ time with linear work, the algorithm runs in the claimed time bound.

4. On-Line Algorithm. We now present the second main result of the paper: an optimal randomized algorithm for the on-line version of the two-dimensional arrangements problem. The on-line problem is to construct an arrangement by inserting lines one at a time. We must complete the insertion of one line before starting the insertion of the next line. We do not know the position of a line before we begin inserting it. The sequential algorithm for constructing an arrangement builds the arrangement one line at a time, so it solves the on-line problem. We give an optimal randomized parallel algorithm for the problem. The algorithm inserts a line into an arrangement of n lines in $O(\log n)$ time using $n/\log n$ processors. The algorithm is for an EREW PRAM. Our algorithm relies on making random choices. The results of the algorithm are always correct, and it succeeds in inserting a line in $c \log n$ time with high probability. Since we are using this algorithm to insert a sequence of n lines, it is important that our performance guarantee is “with

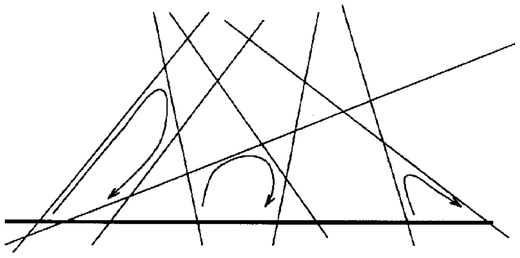

Fig. 6. Inserting a line with independent subtraversals.

high probability," so that we can say it is very likely that all of the insertions succeed within the time bound.

Our on-line algorithm for the problem works along the same lines as the sequential algorithm. The sequential algorithm inserts a line l into an arrangement by traversing the faces adjacent to l. This algorithm works since the number of segments on faces adjacent to l is $O(n)$, and we can move from segment to segment in constant time. For a parallel version of this algorithm, we perform the traversal starting from many intersections along l simultaneously as shown in Figure 6. For a line h in the arrangement, we can locate where the intersection of h and l fits in h's sorted list of intersections in $O(\log n)$ time by binary search. We can then traverse the faces adjacent to l starting from h. Since it takes $O(\log n)$ time to find the starting point for a traversal, we can afford to start from $n/\log n$ points. We choose our starting points by selecting $n/\log n$ lines from the arrangement at random. We would like to perform the subtraversals from each of the starting points, but we run into a major difficulty. The total length of the traversal is $O(n)$, so they have average length $O(\log n)$. However, it is possible that a moderately large number of subtraversals have length greater than $c \log n$, so it is not possible to complete them in $O(\log n)$ time. This difficulty arises for two distinct reasons:

(1) The arrangement may have large faces.
(2) When we select $n/\log n$ elements out of set of size n, some of the gaps between selected elements will be larger than $c \log n$.

The on-line problem is broken into two parts. The first part is to insert a line into the arrangement. Inserting a line l into the arrangement means that we construct a list that contains the intersections of l with the other lines in sorted order, and, for each line h, we insert the intersection point of h and l into h's list of intersections. The second part of the computation is to update the data structure to reflect the addition of the line l. The data structure represents the lines both as a planar subdivision, and as lists of intersections. The subdivision allows the segments bounding a face to be traversed in order. The faces are represented as doubly linked lists of edges. Large faces have binary trees embedded in them which allow us to determine the intersection points of a line with a face in time that is logarithmic in the face size. The other representation of the arrangement gives the intersections along each line in sorted order. This information is used in both a random access fashion as well as to perform binary search. For the purpose of discussion it is best to think that for each line we have an array that gives its intersections with other lines in sorted order. The actual data structure is more complicated, since it must be updated in essentially constant time. We postpone out discussion of the intricacies of the data structure until after our presentation of the algorithm.

Main Idea. We begin with $n/\log n$ starting points for our traversal. This gives us $n/\log n$ tasks to perform with total execution time bounded by cn. We have $n/\log n$ processors available. We could complete all but $\varepsilon n/\log n$ of the tasks by executing each task for $c \log n/\varepsilon$ time. However, we cannot afford to leave this many incomplete tasks (our algorithm will complete all but $n/\log^{3/2} n$ of the tasks). The main idea is to redefine the traversal so that each of the subtraversals can be accelerated by a factor of S by assigning S processors to it. This allows us to transform our problem into one consisting of $n/\log n$ tasks with a total execution time bounded by $c'n/S$ to be scheduled on $n/S \log n$ processors. By using the Cole–Vishkin [9] scheduling algorithm we can complete all but $n/S \log n$ of the tasks in $O(\log n)$ time. After we have completed the subtraversals, there is a substantial cleanup phase in which we identify and place all of the intersections. However, it is redefining the traversal and applying Cole–Vishkin scheduling that is the key to the solution.

Traversal Algorithm. We describe our algorithm as if the line l were horizontal. The algorithm begins by selecting $n \log n$ lines at random from the arrangement, and then computing a sorted list of their intersections with l. For convenience we also select the lines that have the leftmost and rightmost intersections with l. We denote this subset of lines by $\hat{H} = h_1, \ldots, h_m$ and the set of intersection points by $q_1, \ldots, q_m$ where q_i is the intersection of h_i with l and the intersections are ordered $q_1, \ldots, q_m$ from left to right.

The full traversal consists of a set of subtraversals, where the ith subtraversal is a path P_i from q_i to q_{i+1}. The angle between consecutive segments on a subtraversal is less than $180°$, so following the subtraversal from left to right corresponds to a series of right turns. The segments need not arise from adjacent intersections in the arrangement. Thus, if the segment (p_1, p_2) of the line h is a portion of a subtraversal, then h may have intersections with other lines between p_1 and p_2. Figure 7 gives an example of a subtraversal.

We define the *length* of the traversal to be the total number of intersections that are encountered during the construction of its subtraversals. This includes both the inter-

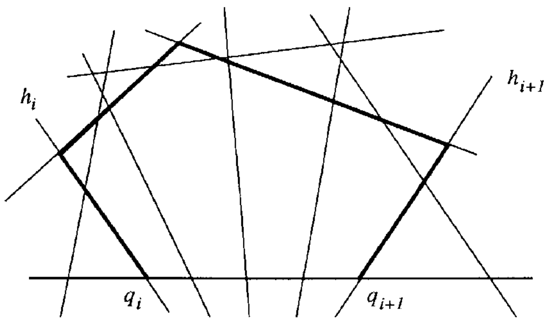

Fig. 7. Subtraversal P_i.

sections that form the endpoints of the segments and intersections that are interior to segments. The total length of the traversal must be bounded by cn for some constant c, so that we can construct it within our resource bound.

In order to have a traversal algorithm that we can efficiently parallelize, we process the intersections along a line in *blocks*. By this we mean that if we are at the intersection v on the line h, we consider the next K intersections along h simultaneously to decide which segment to traverse when we leave h. We assign S processors to the traversal (where $S = \sqrt{K}$), and achieve a speedup of a factor of S over the time for a single processor.

Constructing a Subtraversal. The subtraversal algorithm finds a path P_i from q_i to q_{i+1} for each i. We construct the path P_i by stepping along the lines of the arrangement, turning clockwise at intersections, except when the intersection is with a line that intersects l between q_i and q_{i+1}. We implement this algorithm in a manner that allows us to achieve a limited parallelism on the traversal. Suppose that line h has been identified as being part of the traversal starting from p, the intersection of h and some other line. We need to decide which line follows h in the traversal. We examine the first K intersections along h following p. Our choice of the next line is basically the one that allows us to make as sharp a turn as possible without selecting a line that intersects l between q_i and q_{i+1}. If none of the intersections allows a suitable clockwise turn, then we consider the next K intersections along h. We give a more complete specification of the selection process below. We allocate $S = \sqrt{K}$ processors to this. Assuming that the intersections of h are stored in an appropriate data structure, we can process each block of intersections in $O(S)$ time.

We now give a precise version of the traversal algorithm. The complications in the traversal algorithm arise because we must guarantee that it has length $O(n)$. The subtraversal from q_i to q_{i+1} is divided into a *left subtraversal* and a *right subtraversal*. The left subtraversal begins at q_i and proceeds clockwise until a line is encountered that intersects l to the right of q_{i+1}, and the right subtraversal begins at q_{i+1} and proceeds counterclockwise until a line is encountered that intersects l to the left of q_i. Since the subtraversals are mirror images of each other, we concentrate on the left traversal.

The algorithm for constructing the left subtraversal maintains a current line and a current intersection. We begin with the current line as h_i and the current intersection as q_i. We examine the lines that give rise to the next K intersections along the current line h. There are several cases that can occur:

1. If one of the lines intersects l to the right of or at q_{i+1}, then the left subtraversal is finished. We call the first of these lines the *left boundary*.
2. Case 1 does not occur and one of the lines is a member $\hat{H}$. We choose the first such line h' as the current line, and the current intersection becomes the intersection of h' and h.
3. Cases 1 and 2 do not occur, and one of the lines intersects h_{i+1} "below" the intersection of h and h_{i+1}. We choose the line h' that has the "lowest" intersection with h_{i+1}, as in Figure 8. (The lowest point on h_{i+1} is the intersection of h_{i+1} and l. We consider the line h_{i+1} to wrap around $+\infty$).
4. Cases 1–3 do not occur. We advance the current intersection K intersections along h.

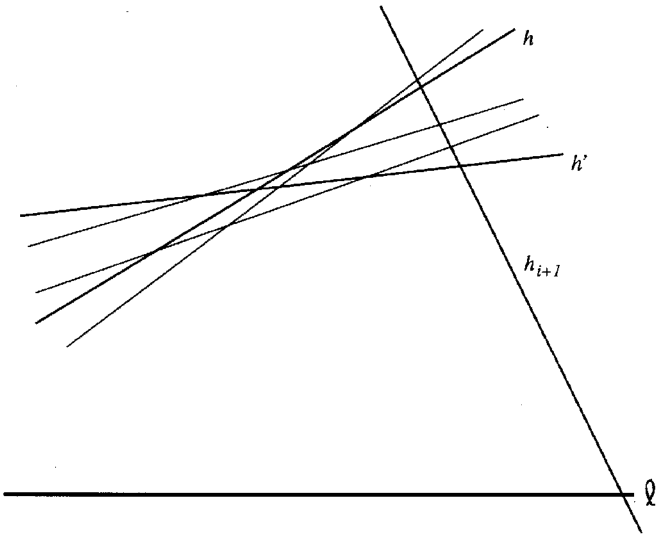

Fig. 8. Selection of the next intersection.

If the left boundary intersects the right subtraversal, then the full subtraversal is formed by following the left subtraversal, then following the left boundary, and then following the right subtraversal (in reverse) from its intersection. If the left boundary does not intersect the right subtraversal, then the lemma below shows that the right boundary intersects the left subtraversal, so that the full subtraversal is formed by following the right subtraversal, then the right boundary, and then the left subtraversal.

LEMMA 4.1. *The left boundary intersects the right subtraversal or the right boundary intersects the left subtraversal.*

PROOF. If the right boundary does not intersect the left subtraversal, then the left boundary intersects the right subtraversal, as shown in Figure 9. □

We choose a block size of $K = \log n$. We now prove that the amount of work done in computing all of the subtraversals is $O(n)$.

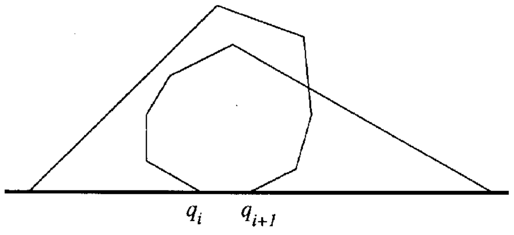

Fig. 9. Left boundary intersecting the right subtraversal.

THEOREM 4.2. *The total amount of work performed during the construction of the subtraversals is $O(n)$.*

The proof that the amount of work is linear has to account for all of the intersections that are examined. Using the same analysis as for the sequential algorithm it is immediate that Case 2 arises $O(n/\log n)$ times, so Case 2 accounts for $O(n)$ intersections in total. The key to the proof is to show that Case 3 accounts for only $O(n)$ intersections. It is possible for a line to be encountered by many of the subtraversals. Our argument is that only a few lines that are encountered during the ith left subtraversal are ever encountered again.

LEMMA 4.3. *Let X be the set of lines involved in Case 3 intersections during the ith left subtraversal. At most $2\log n$ lines from X are involved in Case 3 intersections of subsequent left subtraversals.*

PROOF. Let $b_1, \ldots, b_k$ be the blocks of intersections examined during the construction of the ith left subtraversal. We denote the line traversed when block b_j is examined by g_j. Let X_j be the bundle of lines associated with b_j that intersect h_{i+1} below the intersection of g_j and h_{i+1} (see Figure 10). The key to the proof is that the bundles are ordered in the sense that for $j < j'$, to the right of h_{i+1}, the lines in X_j are above those in $X_{j'}$. It is sufficient to prove this for $j' = j + 1$. Let p be the intersection of g_{j+1} and h_{i+1}. We show that if $g' \in X_j$ and $g'' \in X_{j+1}$, then g' is strictly above g'' on the right-hand side of h_{i+1}. This is true because the slope of g' is greater than the slope of g'' and g' intersects h_{i+1} above p while g'' intersects h_{i+1} below p. This is shown in Figure 11.

If $\sum_{1\le j\le k} |X_j| \le 2\log n$ the lemma is trivial, so suppose that $\sum_{1\le j\le k} |X_j| > 2\log n$. Let j' be the largest index such that $\log n < \sum_{j'\le j\le k} |X_j| \le 2\log n$. Such a j' exists since $|X_j| \le \log n$. We claim that only lines in $X_{j'}, \ldots, X_k$ can be involved in any subsequent intersections. In order to intersect a line $g \in X_j$ for $j < j'$, more than $\log n$ lines must be encountered. Since there are only $\log n$ intersections per block, this means that more than one block would have to be involved. However, the first block that contains

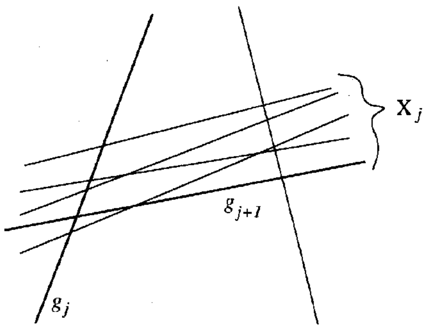

Fig. 10. A bundle of lines.

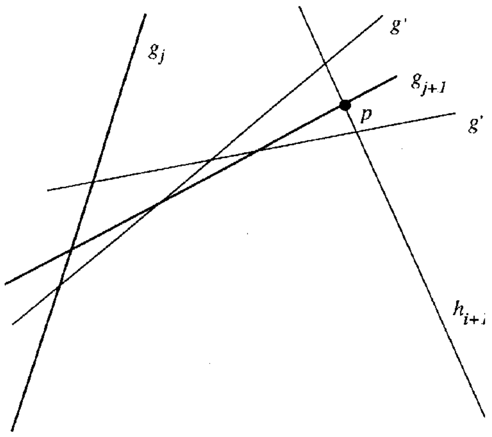

Fig. 11. Ordering of lines.

any of the intersections from $X_{j'}, \ldots, X_k$ would cause the traversal to remain below the line g. □

We can now complete the proof of Theorem 4.2 using an accounting trick. We need to account for the total number of Case 3 intersections. Suppose h' is a line involved in a Case 3 intersection during the ith subtraversal. If this is the first Case 3 intersection for h', we bill it to h', otherwise we bill it to the traversal *of the previous* Case 3 intersection for h'. Each line gets billed at most once, and each traversal gets billed at most $2 \log n$, so the number of intersections in $O(n)$.

Cole–Vishkin Scheduling. The traversal algorithm defines a set of subtraversals between adjacent pairs of points in the set $q_1, \ldots, q_m$. We view each of these subtraversals as a task that we need to execute. It is not necessary to complete all of the tasks within $O(\log n)$ time, since we can have a cleanup phase that processes a small number of the subtraversals by a different method. If we were to assign one processor to a subtraversal of length k, we could complete it in $O(k)$ time. However, if we assign S processors (where S is much smaller than the block size K), we can complete the subtraversal in $O(k/S)$ time. Our approach is to group the processors together in groups of size S referred to as *superprocessors*. This gives us $n/\log n$ tasks to execute on $n/S \log n$ superprocessors, with a total processing requirement of $O(n/S)$. We have $O(\log n)$ time available and want to complete as many tasks as possible. What we have gained by grouping the processors, is that we are now able to load balance; when a processor completes a task, it can then move on to some uncompleted task. A direct application of the Cole–Vishkin deterministic scheduling algorithm [9] says that in $O(\log n)$ time we can execute all of the tasks of size less than $c \log n$. Since tasks of size less than $c \log n$ correspond to subtraversals of length less than $cS \log n$, this allows us to traverse most of the paths. If we are able to complete the subtraversal between q_i and q_{i+1} we say that the gap (q_i, q_{i+1}) is *covered*, and if we are not able to traverse the segment, we say the gap is *uncovered*.

LEMMA 4.4. *If we choose* $S = \log^{1/2} n$, *then after the traversal is complete, there are* $O(n/\log^{3/2} n)$ *uncovered gaps.*

PROOF. Since the sum of the lengths of the subtraversals is at most cn and we complete all paths of length $\log^{3/2} n$ or less, there are at most $cn/\log^{3/2} n$ subtraversals we do not complete, leaving at most $cn/\log^{3/2} n$ uncovered gaps. □

Covered and Uncovered Gaps. We now use the traversal to find $n/\log^{2/3} n$ roughly equally spaced points along l. We run the sequential algorithm independently from each one of these points to complete the insertion of l. We construct a set H of approximately $n/\log^{2/3} n$ lines by assigning lines independently to H with probability $1/\log^{2/3} n$ and including the two lines having the leftmost and rightmost intersections with l. We must locate where l intersects each of the lines of H. This means that, for each $h \in H$, we must determine where the intersection of l and h falls in the list of intersections for h. If l intersects h in a covered gap, we can use information gained during the traversal to locate the traversal quickly, otherwise we must perform a binary search to locate the intersection point. We address both of these cases below. Figure 12 shows lines in H intersecting covered and uncovered gaps.

We actually choose the set H before the execution of the traversal algorithm described above. When we are doing the traversal we can test each intersection we encounter as to whether or not it is with a line in the set H. Suppose we encounter an intersection with the line $h \in H$ when we are traversing between q_i and q_{i+1} and suppose that h intersects l at q which is between q_i and q_{i+1}. We must determine where the intersection q will go in h's list of intersections. On the traversal we encounter an intersection point p of h with some segment of the traversal. The number of intersections between p and q along h is bounded by $\log^{3/2} n$. This means that we could use a binary search to locate where q

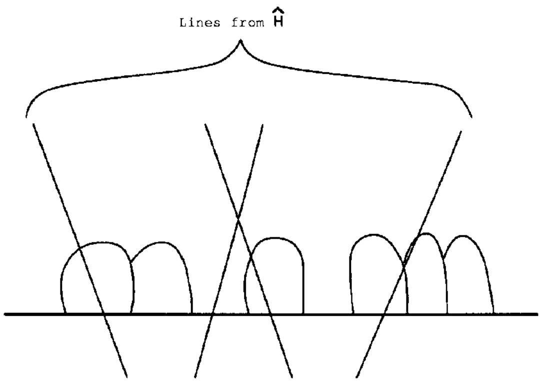

Fig. 12. Covered and uncovered gaps.

belongs in $O(\log \log n)$ time.[3] We can thus place all of the intersections between l and the lines of H that fall in covered gaps within our time and processor bounds.

We handle the intersections that occur in uncovered gaps by binary search. During the traversal we identify all of the lines of H that intersect l in covered gaps, so we are left only with the lines that intersect in uncovered gaps. We perform a binary search on each of these lines to locate the intersection with l. We must show that we do not have too many of these lines that intersect uncovered gaps.

LEMMA 4.5. *The numbers of lines of H that intersect uncovered gaps is bounded by $n/\log n$ with high probability.*

PROOF. The key to the proof is to show that the total size of the uncovered gaps is bounded by $n/\log^{1/3} n$ with high probability. The uncovered gaps arise from first selecting a random set of $n/\log n$ intersections which define all of the gaps, and then performing traversals to determine which are covered or uncovered. To show that the uncovered gaps contain no more than $n/\log^{1/3} n$ intersections, we use the fact that the number of elements in the largest $n/\log^{3/2} n$ gaps is bounded by $n/\log^{1/3} n$ with high probability. The selection of the set H is done by choosing each line with probability $1/\log^{2/3} n$. If we select elements from a set of size at most $n/\log^{1/3} n$ each with probability $1/\log^{2/3} n$, then with high probability, using Chernoff bounds, the resulting set has size bounded by $2n/\log n$. $\square$

After considering both covered and uncovered intervals, we have a set of $n/\log^{2/3} n$ points along l and want to step along the arrangement from each of these points. The one remaining obstacle that we have is that the arrangement may have some large faces. In order to speed up the traversal along large faces, we assume that for each face of the full arrangement we have a balanced binary tree which gives us the segments adjacent to the face in clockwise order. The operation that we perform on a face f is: given the line l and a segment s_1 where l enters f, find the segment s_2 where l leaves f. It is easy to see that we can find s_2 in time logarithmic in the number of segments bounding f using the binary tree. The $n/\log^{2/3} n$ points define a set of $n/\log^{2/3} n$ tasks where a task corresponds to traversing the bounding faces until the next point in the set is found. The total amount of work to execute these tasks is $O(n)$. We can apply the Cole–Vishkin scheduling algorithm to execute all of the tasks that take fewer than $c \log n$ units of work (when we encounter a long task, we just execute it for $c \log n$ units, and then give up if it is not complete). We must now show that almost all of the tasks take less than $c \log n$ time units. We can then have a cleanup phase to take care of the remaining tasks.

LEMMA 4.6. *The amount of work in tasks not completed in $c \log n$ time units is bounded by $2n/\log n$ with high probability.*

[3] The data structure that we give actually supports this operation in $O(\log^{1/6} n)$ time, which is sufficient for our result.

PROOF. There are two things that can cause the tasks to take too long: either the gaps between elements of H can be large, or the faces that are being covered can be too large. We first set aside all of the tasks that have more than $\log^{5/6} n$ intersections with l. It is easy to show that the number of intersections in the gaps that are larger than $\log^{5/6} n$ is bounded by $n/\log n$ with very high probability. If a face is very large (of size n^{ε}), then, even with a binary tree, traversing the face can take more time than we can afford. As long as faces have size less than $K = 2^{\log^{1/6} n}$ time, we can traverse them in time $O(\log^{1/6} n)$. We set aside all tasks that contain faces of size greater than K. Since there are $O(n/K)$ faces of that size, the number we set aside is very small. It follows that the amount of work associated with the tasks we set aside is at most $2n/\log n$. □

5. Data Structures. One of the interesting aspects of this problem is that the data structures that arise are nontrivial. Data structures have not played a major role in the development of parallel algorithms. A review of parallel algorithms shows that in most cases lists and arrays have been sufficient. An explanation of why the data structures are more complicated in this problem than most others is that, since it is an on-line problem, the number of processors $(n/\log n)$ is much smaller than the number of data items that it is necessary to keep track of $(\Omega(n^2))$.

The key to our data structure is to maintain for each line a *sorted* list of intersections. (Our data structure also maintains some geometric information, but this is not a source of difficulty.) For every insertion of a line we must add one intersection to each list. We must do this in $O(\log n)$ time with $n/\log n$ processors. If we only had to worry about the insertion this would not be a difficulty, since we are supplied with an adjacent intersection to each intersection that we add. The difficulty is that we must be able to perform binary search on these lists of intersections. We need to be able to perform binary search on an entire list in $O(\log n)$ time, and need to be able to perform a search between elements separated by $\log^{3/2} n$ elements in time $O(\log^{1/6} n)$. The natural solution is to use some form of balanced tree to represent the list, however, that leads to $\Omega(\log n)$ worst-case time for an insert. We give a data structure that supports the needed operations within our resource bounds. Our data structure is based on a balanced binary tree with a higher branching factor closer to the leaves. An alternate approach would be to adapt the persistent data structures of Driscoll *et al.* [10]. Our data structure is substantially simpler than theirs, since we neither support the full range of operations they describe, nor are our resource requirements as tight.

We represent each of the n lists as a balanced search tree. After we insert a new line into the arrangement, we must update each of these lists by adding one intersection. We have $O(\log n)$ time to perform the update using $n/\log n$ processors. This means the average time for an insert must be $O(1)$. The difficulty is that the rebalancing operations might take $O(\log n)$ time each, which would exceed our resource bounds. Our solution is to give a data structure where only $n/\log n$ of the inserts take $O(\log n)$ time, and the remainder can be done in $O(1)$ time. Our data structure is a balanced binary tree, except that the bottom 12 levels have a branching factor of $O(\log^{1/6} n)$. Figure 13 illustrates our data structure. We refer to each of the subtrees formed by the bottom 12 levels as a *small tree*. Whenever we perform an insert, we just put the item into the appropriate small tree, and rebalance the small tree. This can be done in constant time provided that the set of

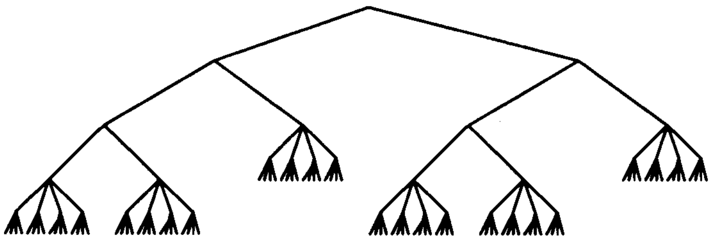

Fig. 13. Balanced tree with higher branching at lower levels.

values in a tree node is represented as a linked list. After every $\log n$th insert into a list, we choose the largest of the small trees in that list, and can rebalance the binary tree above that small tree. The key lemma to prove is that the small trees never get too big.

LEMMA 5.1. *If we are allowed to split a small tree after every* $\log n$ *inserts, then no small tree has more than* $2 \log^2 n$ *leaves*.

PROOF. Whenever we rebalance, we can split one of the small trees into two small trees of half the size. A simplified way of modeling this is to assume we have a set of buckets. We place $\log n$ elements into the buckets, and then get to empty one of the buckets. Since we are attempting to minimize the number of elements in a bucket, we naturally choose the bucket containing the largest number of elements to empty. The question is, after we have inserted n elements, what is the maximum number of elements that a bucket could contain. For every bucket that has $j \log n$ elements, we can identify one bucket containing at least $(j-1) \log n$ elements that we emptied. This means that to get a bucket containing $\log^2 n$ items, we must have emptied $2^{\log n}$ buckets. The detail that we must fix up is that instead of emptying buckets, we split a bucket into two equal-sized buckets. The trick that we use is to assume that we start with n buckets each containing $\log^2 n$ elements. We now argue that after inserting n new elements, we will have no bucket with more than $2 \log^2 n$ items. We can view bucket splitting as emptying, since after a bucket is split, it will contain fewer than $\log^2 n$ elements. □

Since we only rebalance every $\log n$th insert, the work done to insert a line is $O(n)$, so it can be done in $O(\log n)$ time with $n/\log n$ processors.

Each insertion into a small tree is done in constant time. The small trees are B-trees with interior nodes having degree between $\frac{1}{2}\log^{1/6} n$ and $\log^{1/6} n$. Each node in the tree is represented by a doubly linked list. When we insert an element, we are given a pointer to a neighboring element, so it takes constant time to splice the element into the list. When we insert an element, we might have to split a node in the tree. We split a node by maintaining a pointer to the center element in the node. Since the height of the small trees is bounded by a constant, we can rebalance in constant time. We can maintain pointers to the central element of a node by an amortized computation. We only need the central pointer when a node has size $\log^{1/6} n$, so after a node is split, we have $\frac{1}{2}\log^{1/6} n$ inserts before the next split, so we compute the central pointer by traversing the nodes one step at each insert.

It is straightforward to perform binary search in $O(\log n)$ on these trees. If we make sure the small trees have close to $\log^{1/6} n$ elements per internal node, then it is straightforward to do a binary search between two elements separated by $O(\log^2 n)$ elements in $O(\log^{1/6} n)$ time. It is also important to have the internal nodes close to full so that the traversal can be done in blocks.

6. Deletion.

The problem of deleting a line from an arrangement is much easier than insertion. We delete a line by traversing the adjacent faces, starting from $n/\log n$ equally spaced intersections. Since we are given the line, all the difficulties of finding these starting points are avoided. We augment the data structure described above to handle deletes in a straightforward manner. We do not rebalance the trees for deletes. If we have m lines in the arrangement and a total of n inserts have been executed, a line can be inserted or deleted in time $O(\log n)$ using $m/\log n$ processors.

There is room for improvement in this result, since we would like the cost of operations only to depend on the number of elements in the arrangement, and not on the history. We leave open the problem of performing insertion and deletion on an arrangement of size m in time $O(\log m)$ using $m/\log m$ processors.

7. Conclusions.

We have demonstrated a very efficient parallel algorithm for the problem of computing arrangements of line d dimensions. This algorithm runs on a CREW PRAM in near-optimal time and total work. We have also shown an EREW PRAM algorithm for an on-line version of the two-dimensional arrangements problem that is randomized and operates in asymptotically optimal time and total work. This latter algorithm shows some of the interesting problems that arise when dealing with parallel algorithms for on-line problems, particularly in the need for nontrivial data structures.

References

[1] A. Aggarwal, B. Chazelle, L. Guibas, C. O'Dunlaing, and C. K. Yap. Parallel computational geometry. *Algorithmica*, 3:293–326, 1988.

[2] A. Aggarwal and J. Wein. Computational geometry: lecture notes for 18.409, spring 1988. Technical Report MIT/LCS/RSS 3, MIT Laboratory for Computer Science, 1988.

[3] R. J. Anderson, P. Beame, and E. Brisson. Parallel algorithms for arrangements. *Proceedings of the Second Annual Symposium on Parallel Algorithms and Architectures*, 1990, pp. 298–306.

[4] D. Arnon, G. Collins, and S. McCallum. Cylindrical algebraic decomposition, *I* and *II*. *SIAM Journal on Computing*, 13(4):865–889, 1984.

[5] M. J. Atallah, R. Cole, and M. T. Goodrich. Cascading divide-and-conquer: a technique for designing parallel algorithms. *SIAM Journal on Computing*, 18:499–532, 1989.

[6] J. Canny. A new algebraic method for robot motion planning and real geometry. *Proceedings of the* 28*th Symposium on Foundations of Computer Science*, 1987, pp. 29–38.

[7] B. Chazelle. Intersecting is easier than sorting. *Proceedings of the* 16*th ACM Symposium on Theory of Computation*, 1984, pp. 125–134.

[8] B. Chazelle, L. J. Guibas, and D. T. Lee. The power of geometric duality. *BIT*, 25:76–90, 1985.

[9] R. Cole and U. Vishkin. Approximate parallel scheduling. Part I: The basic technique with applications to optimal parallel list ranking in logarithmic time. *SIAM Journal on Computing*, 17:128–142, 1988.

[10] J. Driscoll, N. Sarnak, D. Sleator, and R. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38:86–124, 1989.

[11] H. Edelsbrunner. *Algorithms in Combinatorial Geometry*. Springer-Verlag, New York, 1987.

[12] H. Edelsbrunner, J. O'Rourke, and R. Seidel. Constructing arrangements of lines and hyperplanes with applications. *SIAM Journal on Computing*, 15(2):341–363, 1986.

[13] M. T. Goodrich. Intersecting line segments in parallel with an output-sensitive number of processors. *Proceedings of the First Annual Symposium on Parallel Algorithms and Architectures*, 1989, pp. 127–136.

[14] M. T. Goodrich. Constructing arrangements optimal in parallel. *Proceedings of the Third Annual Symposium on Parallel Algorithms and Architectures*, 1991, pp. 169–179. Also in *Discrete & Computational Geometry*, 9:371–385, 1993.

[15] T. Hagerup, H. Jung, and E. Welzl. Efficient parallel computation of arrangements of hyperplanes in d dimensions. *Proceedings of the Second Annual Symposium on Parallel Algorithms and Architectures*, 1990, pp. 290–297.

[16] J. JáJá. *An Introduction to Parallel Algorithms*. Addison-Wesley, Reading, MA, 1992.

[17] M. McKenna. Worst-case optimal hidden-surface removal. *ACM Transactions on Graphics*, 6(1):19–28, 1987.

[18] J. H. Reif and S. Sen. Polling: a new randomized sampling technique for computational geometry. *Proceedings of the* 21*st ACM Symposium on Theory of Computation*, 1989, pp. 394–404.

[19] A. Tarski. *A Decision Method for Elementary Algebra and Geometry*. University of California Press, Berkeley, CA, 1951.